Alice and Bob Meet Equational Theories

David Basin, Michel Keller, Saša Radomirović, and Ralf Sasse

Institute of Information Security, Department of Computer Science, ETH Zurich

Dedicated to José Meseguer on his 65th Birthday.

Abstract. Cryptographic protocols are the backbone of secure communication over open networks and their correctness is therefore crucial. Tool-supported formal analysis of cryptographic protocol designs increases our confidence that these protocols achieve their intended security guarantees. We propose a method to automatically translate textbook style Alice&Bob protocol specifications into a format amenable to formal verification using existing tools. Our translation supports specification modulo equational theories, which enables the faithful representation of algebraic properties of a large class of cryptographic operators.

1 Introduction

Internet security builds on cryptographic protocols that achieve properties such as secrecy, entity authentication, and privacy. The correct operation of these protocols is critical and manual analysis is not up to the task. Indeed, some protocols were used for years before flaws were detected using symbolic analysis tools [19,4]. Today, there are many such tools available based on different formalisms and specification languages: ProVerif [6] uses the applied pi calculus, Scyther [13] uses role scripts, Maude-NPA [15] uses strands [17], and Tamarin [29] specifies protocols using multiset rewriting. Unfortunately the input languages of all of these tools are difficult for non-expert users to master, which hinders the widespread acceptance and use of these tools.

The starting point for this paper is our work on the Tamarin tool, which has been used successfully to analyze many cryptographic protocols [25,30,5]. Tamarin uses multiset rewriting as its input language, which is very general; but this generality makes it difficult to use, especially for non-experts. Hence an attractive proposition is to support, additionally, a simpler and more intuitive language, closer to the text-book notation that many users know from their studies. The most popular language of this kind is generally known as Alice&Bob protocol specifications. Due to its popularity, versions of it have been considered for other analysis tools [1]. It is indeed simple, but it suffers from ambiguities and imprecision.

We propose a new way to specify protocols in an Alice&Bob style that supports specification *modulo user-specifiable equational theories*. This enables one to specify, along with a protocol, the algebraic properties of the cryptographic operators used. To support this, we analyze the protocol specification's executability and translate executable specifications into an intermediate language based

on role scripts. We then determine which checks should be made on the messages by the participants, based on their current knowledge and the equational theory, and insert those checks into the compiled role scripts. Non-executable specifications are rejected, and warnings to the user are displayed when checks cannot be inserted as expected, for example, when the same name is used repeatedly, but due to encryption the principal cannot verify if all occurrences are instantiated in the same way. The role scripts can then be further translated to any protocol analysis tool's input language, as our theoretical results are general. We have implemented this first general-purpose translation step and, for Tamarin's input language, we have also implemented the second step. Taken together, this provides an automatic translation from Alice&Bob specifications to the Tamarin tool's input language [18].

As mentioned, equational theories are used to model the algebraic properties of cryptographic operators. Support for different equational theories therefore allows a more precise analysis of protocols. Specifying suitable equational theories is fundamental for the symbolic analysis of cryptographic protocols as otherwise attacks may be missed. Moreover, for some protocols, their execution would even be impossible without algebraic properties. For example, for Diffie-Hellman key exchange, the partners cannot establish the same key without exponentiation being commutative. We focus on subterm-convergent theories, which we require for the results presented in this paper. Note that these theories have the finite variant property [12,16], which is a prerequisite for many automated protocol analysis tools as well.

To handle the imprecision inherent in basic Alice&Bob notation, we build upon previous work that makes explicit many of this notation's assumptions. In particular, Caleiro, Viganò, and Basin [9] provide an operational semantics based on the spi calculus that formalizes how principals construct and parse messages and makes explicit what checks should be made by honest principals. Mödersheim [26] studies a similar problem in the context of equational theories. See Section 5 for a more detailed comparison.

We also build upon and take inspiration in our work from the research and tools of José Meseguer. Tamarin uses Maude [11] as a back-end for unification modulo equational theories. Moreover, inspired by Maude-NPA [15], Tamarin computes variants following [16]. This motivated the design decision of supporting user-specified subterm-convergent equational theories in addition to built-in theories with the finite variant property that is used both in Tamarin and in the translation we present in this paper.

In Section 2 we describe Alice&Bob notation. In Section 3 we present the role-script notation for protocols, explain how to decide the executability of given Alice&Bob protocols and how to add appropriate checks, and provide an example. We describe our automated translation to Tamarin in Section 4 and compare to related work in Section 5, before we draw conclusions in Section 6.

1. $C \to S$: C, V, n
2. $S \to C$: $\{k, V, n\}_{k_{CS}}, \{k, C\}_{k_{VS}}$
3. $C \to V$: $\{t, t'\}_k, \{k, C\}_{k_{VS}}$
4. $V \to C$: $\{t\}_k$

Fig. 1. An intuitive but ambiguous description of an authentication protocol.

2 Alice&Bob Protocol Notation

In this section we formalize Alice&Bob notation. First, we highlight ambiguities with the text-book version of this notation due to its inherent imprecision, and we explain the general idea of our formalization. Afterwards we specify this notation in more detail.

2.1 Overview

In Alice&Bob notation, a protocol is specified as a list of *message exchange steps* of the form

$$A \to B\colon\ msg.$$

These steps describe the actions that are performed by honest principals in a protocol run. The semantics of an Alice&Bob specification defines the behavior of the principals running the protocol. Our semantics is based on the work of Caleiro, Viganò, and Basin [9].

To illustrate the need for a formal semantics, consider the simplified basic Kerberos authentication protocol [27] shown in Figure 1. At first glance, this protocol's meaning seems clear. The principal in role C sends his identity, the name of a resource V, and the nonce n to the authentication server S. A *nonce* is an arbitrary number to be used only once in a security protocol. The principal in role S then responds by returning two ciphertexts, the first one generated with a shared key k_{CS} between C and S and the second generated with a key k_{VS}, shared between V and S. The principal C then decrypts the first ciphertext to obtain the key k and verifies the nonce n and the intended communication partner V. If these checks succeed, then C encrypts fresh nonces t and t' under k and sends this ciphertext along with the second ciphertext to V. The principal V then responds with the encryption of t under the key k, which is obtained by decrypting the second ciphertext $\{k, C\}_{k_{VS}}$.

While this account provides a high-level explanation of the protocol's workings, the precise actions the principals must take are not fully spelled out. In order to send the message $\{k, V, n\}_{k_{CS}}, \{k, C\}_{k_{VS}}$ in Step 2, S must first construct it. Intuitively, one would assume that S knows k_{CS} and k_{VS} and generates a fresh key k and can therefore construct the message. But this is not stated explicitly. It is possible that the protocol's designer had in mind that k is known

only to C and V, while S only knows the two ciphertexts in Step 2. The specification as given does not resolve this ambiguity. It is also unclear whether n is actually a nonce, even though the choice of name suggests this. It could just as well be a constant or a publicly known value, neither of which we would assume of a nonce.

Another aspect left implicit in Figure 1 is what the principals do with the messages they receive. If we assume that k_{CS} actually denotes the shared secret key between C and S and that both know the key and C has no other prior knowledge, then C should extract the key k by decrypting the first ciphertext of Step 2 with the secret key k_{CS}. This is the only way C can construct the message $\{t, t'\}_k$ in Step 3. For this reason, we must formalize the new information gained by analyzing incoming messages based on the knowledge that a principal has.

We formalize Alice&Bob notation based on the notion of knowledge, given by a set of messages, which grows during protocol execution when new messages are received or fresh nonces are generated. We define what information is stored, how incoming messages are parsed and compared to existing knowledge, and how messages are composed for sending.

In the rest of this section, we provide a complete formalization of Alice&Bob notation, which is the basis of our Alice&Bob input language and our translation to the intermediate representation format that follows. Note that Alice&Bob notation is independent of the adversary model. We do not define the capabilities of the adversary as they can be independently specified.

2.2 Messages and Message Model

We use a signature Σ containing three sorts, $Fresh$, $Public$, and Msg, where both $Fresh$ and $Public$ are subsorts of Msg. Each operator $f\colon s_1 \dots s_n \to s$ defined on any of the sorts has a top sort overloading, i.e., $f\colon Msg \dots Msg \to Msg$. We assume disjoint sets of countably infinite variables X_s for each sort, with $X = \bigcup_s X_s$. $\mathcal{T}_{\Sigma,s}$ is the set of ground terms of sort s and $\mathcal{T}_{\Sigma,s}(X)$ is the set of terms of sort s. We use $\mathcal{T}_\Sigma$ and $\mathcal{T}_\Sigma(X)$ for the corresponding term algebras.

By default, Σ includes the following operators: pairing, projections (first or second element of a pair), symmetric and asymmetric encryption and decryption, digital signing, and hashing. The set of equations $\mathcal{E}$ defining these operators gives rise to an equational theory $(\Sigma, \mathcal{E})$.

In addition to the default operators, users can specify further operators together with the equations defining them. The user-defined equations must be subterm-convergent: directed from left to right, the resulting rewrite system must be convergent and the right-hand side of any equation is either (1) a constant that is in normal form with respect to the rewrite system or (2) a strict subterm of the left-hand side. Combining the default theory with such additional user-supplied operators and their equational specification yields a theory that is also subterm-convergent. We say that a term t is *derivable* from a set of terms M if it can be constructed by repeated application of operators in Σ to the terms in M under the equational theory $\mathcal{E}$.

We define positions in terms as sequences $p = [i_1, \ldots, i_n]$ of positive natural numbers. We use $t|_p$ to denote the subterm of t at position p, and for the empty sequence $[\,]$ we define $t = t|_{[\,]}$. We use the operator $\cdot$ to concatenate sequences. Two positions p, p' are siblings if $|p| = |p'|$ and there is an immediate parent position p'' such that $p = p'' \cdot [i]$ and $p' = p'' \cdot [i']$ with $i \neq i'$ for two natural numbers i and i'.

All other standard notation follows the account of Baader and Nipkow [2].

2.3 Alice&Bob in Detail

Prior to defining Alice&Bob notation, we first describe the principal's initial knowledge and the knowledge that a principal has during a protocol's execution

Knowledge and Basic Sets. Principals remember messages they acquire during protocol execution. We call the set of messages that are derivable from acquired messages the principal's *knowledge*. Knowledge is essential for constructing messages to be sent, analyzing received messages, and comparing the messages received in a protocol step to messages received in an earlier step.

A principal's knowledge is in general infinite; if Alice knows a message m, she can immediately derive infinitely many messages, for example by concatenating arbitrarily many copies of m. To finitely represent the knowledge of the principals participating in a protocol, *basic sets* proposed by Caleiro et al. [9] can be used. A basic set for M is a minimal set of terms from which all terms in M can be derived.

Initial Knowledge. A principal's initial knowledge is the basic set of messages that the principal knows prior to performing any protocol actions, that is, before the first sending or receiving action. A default initial knowledge can be generated from the protocol's context, i.e., the protocol and all the roles appearing in it, or the initial knowledge can be explicitly given.

Alice&Bob Protocol Specifications. We now define Alice&Bob protocol specifications.

Definition 1. *An* Alice&Bob protocol specification *is a quadruple* $(Spec, \rho, \Sigma, \mathcal{E})$*, where:*

- *Spec is a finite sequence* $step_1, \ldots, step_n$ *of message exchange steps where, for* $t \in \{1, \ldots, n\}$*,* $step_t$*, has the form*

$$\mathbf{label_t}.\quad S \to R\colon\ (n_1, \ldots, n_v).m\,.$$

  Here R *and* S *are distinct role names (terms of sort* $Public$*),* $n_1, \ldots, n_v$ *are distinct variable names of sort* $Fresh$*,* $\mathbf{label_t}$ *is a* unique *name given to this message exchange step, and* $m \in \mathcal{T}_\Sigma(X)$ *is a message.*

- *ρ is a partial map $\mathcal{T}_{\Sigma,Public}(X) \to \mathcal{P}(\mathcal{T}_{\Sigma}(X))$ from role names to sets of messages representing that role's explicit initial knowledge in the protocol. Note that this explicit initial knowledge may very well be empty and can then be omitted. The notational conventions below explain the standard initial knowledge that is always assumed to be available to each role.*
- *$(\Sigma, \mathcal{E})$ is a subterm-convergent equational theory specifying operators and their defining equations.*

Note that we require fresh nonces $(n_1, \ldots, n_v)$ to be stated explicitly. They are assumed to be generated randomly by the sender at the beginning of the step, before the message is constructed and sent. This information is actually redundant: since we know the initial knowledge, we can find out if a nonce is fresh. Nevertheless, we explicitly declare fresh nonces to improve readability and to help catch specification errors.

Our definition says that the fresh nonces must have *distinct* variable names. This not only includes the current message exchange step, but also the complete protocol. Two fresh variables with the same name may not appear in a protocol and a variable that appears in the initial knowledge of a principal may not be redefined as a fresh variable. Moreover, a fresh variable name must not coincide with any role name; this is ensured by having different sorts for role names and fresh variables.

The labels are given just for reference and we omit them in many cases. Also, we drop the parentheses enclosing the fresh variable names when there are none.

Notational Conventions. Alice&Bob protocol specifications rely heavily on implicit notational conventions. For instance, the notation k_{CS} suggests that this term is a shared key between C and S. There is also no need to mention that C is the client and S is the server. We need to be a bit more formal in a computer-interpretable input language, but we also use the following notational conventions and some short-hands to keep our Alice&Bob notation compact yet still precise:

- Variables representing fresh terms (of sort $Fresh$) and general message terms (of sort Msg) are denoted by lower case letters, possibly with subscripts.
- Variables representing public terms (of sort $Public$), including role names, are denoted by capital letters. In the following example, the principal in role A sends her own name to the principal in role B,

  $$A \to B\colon\ A.$$

  Note that the 'A' before the colon denotes the name of a role while the 'A' after the colon denotes the name of the principal that executes role A during an execution of the protocol.
- Constants are public terms that are denoted as strings in single quotes. Below, the principal in role A sends the constant 'Hello!' to the principal in role B:

$$A \to B\colon \text{ 'Hello!'}.$$

- The asymmetric encryption of a message m with the public key $pk(A)$ of principal A is denoted by $\{m\}_{pk(A)}$. This is syntactic sugar for $\mathtt{enc}(m, \mathtt{pk}(k))$, where $\mathtt{enc}(\_,\_)$, $\mathtt{dec}(\_,\_)$, and $\mathtt{pk}(\_)$ are operators defined by the equation

$$\mathtt{dec}(\mathtt{enc}(m, \mathtt{pk}(k)), k) = m. \tag{1}$$

  Thus $pk(A)$ is syntactic sugar for $\mathtt{pk}(k)$, where k is a private key (a fresh term) that A knows.
- Digital signatures are a special case of asymmetric encryption, with $sk(A)$ denoting the secret key k of the principal in role A with the associated public key denoted as $pk(A)$. Signature verification is defined by the equation

$$\mathtt{sigverify}(\mathtt{sign}(m, sk(A)), m, pk(A)) = \mathtt{True}. \tag{2}$$

  This equation is treated as a predicate by principals. In protocol analysis tools such as Tamarin, it is possible to restrict the protocol executions analyzed to those that fulfill the predicate.
- The symmetric encryption of a message m with the secret key $k(A, B)$ shared by the principals A and B is denoted by $\{m\}_{k(A,B)}$. This is syntactic sugar for $\mathtt{senc}(m, k)$, where $\mathtt{senc}(\_,\_)$ and $\mathtt{sdec}(\_,\_)$ are operators defined by the equation

$$\mathtt{sdec}(\mathtt{senc}(m, k), k) = m \tag{3}$$

  and $k(A, B) = k$ for a shared secret key k (a fresh term) that both A and B know.
- For each principal A, we assume that A's initial knowledge contains its own private key $sk(A)$. Moreover, A's initial knowledge includes for each principal B that principal's public key $pk(B)$, and the shared secret key $k(A, B)$. These keys need not be explicitly specified as a principal's initial knowledge. The corresponding encryption functions and their definitions (1) and (3) are included in the equational theory by default. Similarly, pairing and projection of terms are also included and they satisfy the following two equations.

$$\mathtt{fst}(\mathtt{pair}(a, b)) = a$$
$$\mathtt{snd}(\mathtt{pair}(a, b)) = b$$

  For $n \geq 2$, we write $(a_1, \ldots, a_n)$ for the repeated, left-associative application of the pairing operator to the terms $a_1, \ldots, a_n$. We drop the parentheses whenever the resulting expression is unambiguous.

With these conventions, Figure 2 shows a specification of the simplified Kerberos authentication protocol in our Alice&Bob syntax.

3 From Equations to Rewriting Rules

We translate Alice&Bob protocol scripts to a tool's input language via an intermediate representation called *role scripts*. A role script represents a principal's

1. $C \to S$: $(n).\ C, V, n$
2. $S \to C$: $(k).\ \{k, V, n\}_{k(C,S)}, \{k, C\}_{k(V,S)}$
3. $C \to V$: $(t, t').\ \{t, t'\}_k, \{k, C\}_{k(V,S)}$
4. $V \to C$: $\{t\}_k$

Fig. 2. The simplified Kerberos authentication protocol in our Alice&Bob specification.

view of the protocol specification. It consists of the principal's send and receive actions, which each take a message as an argument. The principal sends and receives messages to and from a channel, without information on who the actual communication partner is.

3.1 Role Scripts for Protocols

Given input in Alice&Bob notation, we first check its executability under the fine interpretation of Caleiro et al. [9]. We explain this in Section 3.2. Then we translate the protocol to role scripts, one role script for each protocol role. We add appropriate checks to be taken by the principals receiving messages, in Section 3.3. In Section 3.4 we provide an example that illustrates our algorithms. Afterwards we can generate output in the input language of any suitable protocol verification tool, and we have implemented this for TAMARIN. Output for other tools could be generated from the intermediate role scripts in a similar fashion.

Role scripts have the same syntax as Alice&Bob messages described in Section 2.2. They also include the knowledge principals acquire from the messages they receive; in this way, the role scripts make all information explicit.

We represent a protocol by the equational theory its operators support, the security goals of interest, and a collection of role scripts that specify the message exchanges. Each role script consists of a name, an initial knowledge, and an (ordered) list of actions to be performed. The actions are sending or receiving a message, creating nonces, and updating the principal's knowledge afterwards. Both incoming and outgoing messages contain the name of the designated partner role. In the derived role script specification, our algorithm explicitly states which generated names are fresh, i.e., nonces, and the checks that need to be performed by roles on received messages. Moreover, the specification of the secrecy of a given term, as well as non-injective and injective agreement [20] between roles is supported.

3.2 Deciding Executability

An Alice&Bob specification is a list of message exchange steps. A step is *executable* if the sender's knowledge (at that point in the protocol's execution) is sufficient to create the message specified to be sent. Message creation here refers to the capability of the principal to (i) generate new nonces and add them to his

knowledge, and (ii) apply operators to messages in his knowledge. Note that to derive a principal's knowledge based on the messages he previously received, it is necessary to apply operators (like decryption) and use their algebraic properties. An Alice&Bob specification is executable if all steps of all roles are executable. Thus to decide whether an Alice&Bob specification is executable, we must determine for each step of every role whether the term specified to be sent in the step can be derived from the principal's knowledge at that point in the protocol.

Without equational theories, we have a simple separation of knowledge derivation rules into construction and deconstruction rules, reminiscent of Paulson's inductive approach, where they are called synthesis and analysis rules. A similar procedure can be used with equational theories that are subterm-convergent and we now give the high-level description of this. All operators can be applied as *construction* rules that produce, bottom-up, constructed terms. From each equation of the theory, we extract several *deconstruction* rules that produce, top-down, (de-)constructed terms.

To prevent endless loops, we split an agent's knowledge into two sets, the set of constructed terms $\mathcal{C}$ and the set of terms to be deconstructed $\mathcal{D}$. Construction rules may only be applied to terms in the set $\mathcal{C}$ and produce terms that we add to $\mathcal{C}$. Deconstruction rules take a term from the set $\mathcal{D}$ and zero or more terms from $\mathcal{C}$ and produce terms that we add to $\mathcal{D}$. Moreover, we have one rule that may add any term from $\mathcal{D}$ to $\mathcal{C}$. As a result, constructed terms never need be deconstructed later because deconstruction yields a subterm (due to the subterm-convergence property) that must have been used in the construction of the term, and thus is known already and this shorter derivation can be used.

More precisely, construction rules are created as follows: Let $\mathcal{C}^n$ denote the n-fold Cartesian product of $\mathcal{C}$. Let $f \in \Sigma$ be an n-ary function symbol. Then we add the rule $(t_1, t_2, \ldots, t_n) \in \mathcal{C}^n \vdash f(t_1, \ldots, t_n) \in \mathcal{C}$ to the set of construction rules. In particular, there is exactly one construction rule for each function symbol.

For deconstruction rules, let $l = r$ be an equation oriented such that r is a constant or a strict subterm of l. If r is ground, no deconstruction rule is necessary, as the specifications of ground terms are public knowledge. Otherwise, we obtain a set of deconstruction rules for every occurrence of r in l as follows. (See *drules* and *cprems* functions in [28, Section 3.2.3].)

1. Consider the set *positions* of all positions p in l that mark a subterm equal to r, i.e., all p such that $l|_p = r$.
2. For each $p \in$ *positions*, consider all positions D_p that are strictly above p and not equal to $[\,]$, i.e., subterms $l|_{p'}$ of l that contain the term r at position p'' in $l|_{p'}$, i.e., $l|_{p'.p''} = r$ for some $p'' \neq [\,]$, except for $l|_p$ and l.
3. For each $p' \in D_p$, consider the set $C_{p'}$ of positions that have a sibling above or equal to p'.
4. The deconstruction rules are

$$l|_{p'} \in \mathcal{D} \wedge \left(\bigwedge_{q \in C_{p'}} l|_q \in \mathcal{C} \right) \vdash l|_p \in \mathcal{D}, \tag{4}$$

```
Input: M.
M' := M
(*) ∀m ∈ M, ∀drule(l, q) ∈ drules:
        t := apply drule(l, q) to m
            if t ≠ ⊥ ∧ t ∉ M
                then M' := M' ∪ {t}
if M' ≠ M then M := M' goto (*)
return M
```

```
Input: M, t.
D := close(M)
C := M ∪ D
if t ∈ constructall(C, size(t))
    then return true
    else return false
```

Fig. 3. Algorithm `close`(M) (left) and algorithm `derivable`(M, t) (right).

for each $p' \in D_p$, where $l|_{p'}$ must be a term in $\mathcal{D}$ and the terms corresponding to positions in $C_{p'}$ are in $\mathcal{C}$. The deconstructed term $l|_p$ is added to $\mathcal{D}$.

In the following, we call these rules *drules*, and we denote such a rule as $drule(l|_{p'}, p'')$, where p'' is as in Step 2 above and $l|_{p'}$ is the left-most term in Rule (4). By applying a drule $drule(x, q)$ to a term t we obtain $t|_q$ if x matches t and there exist terms in $\mathcal{C}$ fulfilling the remaining requirements of the left-hand side of the rule, and $\perp$ otherwise. We can now use these rules with the following effective, but computationally expensive, procedure to compute the closure of $\mathcal{D}$ under application of drules to terms in $\mathcal{D}$. The closure algorithm is as follows.

Closure algorithm. Figure 3 (left) provides pseudo-code for the algorithm `close`(M). This algorithm repeatedly applies all drules to all terms in M, adding the resulting terms to M, until a fixed point is reached.

This algorithm always terminates on finite input sets $\mathcal{D}$. Let $\overline{\mathcal{D}}$ be the set of all (sub-)terms of elements in $\mathcal{D}$. Let $\widehat{\mathcal{D}}$ be the result of applying the closure algorithm to $\mathcal{D}$. All terms derivable from $\mathcal{D}$ are in $\overline{\mathcal{D}}$ due to subterm-convergence, so $\mathcal{D} \subseteq \widehat{\mathcal{D}} \subseteq \overline{\mathcal{D}}$. The closure algorithm monotonically increases its set of derived terms M. It terminates when no element is added to M in an iteration. As the resulting set $\widehat{\mathcal{D}}$ is bounded above by the finite set of (sub-)terms, i.e., $\overline{\mathcal{D}}$, termination is ensured.

Derivability. To decide whether a term t is derivable from a given knowledge set M, we use the algorithm `derivable`(M, t), given in Figure 3 (right). The algorithm first computes the closure of M and the size of t. It then uses the procedure `constructall`, with the closure and t's size as input, to generate all terms, up to the given size, which are built from elements of the closure with any operator application. The term t is then derivable if and only if it is in this set of constructed terms. Since t is finite, the derivability check terminates.

Our derivability algorithm is sound: the above derivability procedure only returns `true` for terms that are actually derivable because it only uses drules (in the closure algorithm) and construction rules on the initial set of terms. Our algorithm is also complete as it returns the correct answer `true` for all derivable terms. We sketch a proof by contradiction of this: Assume there is a term t that is derivable from M for which our algorithm returns `false`. That term is derived using operator application and simplification with the equational theory on the terms in M. If t is built using only operator application, then it would trivially

be derivable according to our algorithm, which is a contradiction. Otherwise an equation in the equational theory must be used in the derivation. Pick the smallest subterm s of t on which some equation is applicable, i.e., there is no proper subterm of s for which any equation can be applied. As the equations are subterm-convergent, applying an equation results in a subterm $s|_p$ at some position p (or a constant, which is known anyway) which must be constructed using only operator applications. Then we can replace s by $s|_p$ in the derived term t to create a term t'. Now there is one less equation application possible on t'. We can repeat this until no more equation can be applied as there is a finite number of equation applications to start with. Thus our algorithm returns `true` for this term t as well, which is a contradiction.

3.3 Checking Received Messages

For cryptographic protocols, it is not only important that all participants can generate all the outgoing messages, but also that each participant checks each incoming message as thoroughly as possible, to make sure it is as expected and that the other principals have not deviated from the protocol.

An obvious example of what to check is that a message authentication code received matches the intended message. Similarly, if a principal generates a nonce, sends it, and expects to receive it in a subsequent message, he should check that the nonce he receives actually matches the nonce he generated. These are sensible checks that should be included in principals' role scripts; but one must take care to avoid unrealistic or even infeasible checks.

Example 1. Consider again the Kerberos authentication protocol shown in Figure 2. In this protocol, it is not possible for the agent C to verify in Step 2 that the key k appearing in the first ciphertext is equal to the key k in the second ciphertext. We therefore do not add such a check to C. However, C can verify that the nonce n generated and sent to S as well as the identity of the intended communication partner V in Step 1 are equal to the nonce n and the identity V appearing in the first ciphertext in Step 2. Hence we add these two checks to C.

In general, whenever the same name appears multiple times in a role script, all its instances must be identical, except for those that the principal in question cannot actually analyze and check. An example of this is a key inside a ticket created for a different principal, such as is the case for the principal C in the second ciphertext in Step 2 of Kerberos. The principal cannot see the content of that ticket and thus one must not add checks to this principal on this opaque data.

The key ideas of the algorithms, which we describe in detail in the rest of this section, are as follows. We compute the closure of the agent's knowledge with the above closure algorithm, but track additional information on how terms are derived. Then, for each received message, we check if any of the *accessible* parts of the received message were previously known to the agent. If so, we generate a check that compares the received message part in question to the

previously known terms. To determine which parts are accessible, we use the closure algorithm as well. Essentially, we store all the received terms and the terms derived from them in a marked set. We test if any term pattern in the marked set can be generated in different ways from the current knowledge (which includes other received messages), and if so we generate a check that the resulting role script should make. Note that this includes the possibility of comparing (parts of) two received messages with each other if they are supposed to be the same. We now delve into the formal details.

Let M be the agent's knowledge. We annotate every term in M with its provenance and we distinguish between $\mathcal{D}$-terms and $\mathcal{C}$-terms with different annotations. We can thus partition M into the two subsets $\mathcal{D}$ and $\mathcal{C}$. The annotation is constructed as follows. If $m \in M$ was received in the i-th protocol step, it is in $\mathcal{D}$ and annotated with $[i]$ and denoted $m^{[i]}$. The subterms of a term $m^{[i]}$ are additionally annotated with their position in $m^{[i]}$, while for the term itself we use $m^{[i]}$ as shorthand for $m^{[i:[\,]]}$. We call these annotations *locators*. These terms are obtained by applying a drule. For example, suppose that $m^{[3]} = \langle t, t \rangle$, then the two subterms of $m^{[3]}$ are $m^{[3:[1]]} = t$, $m^{[3:[2]]} = t$. Terms in $\mathcal{C}$ are annotated by sets of locators rather than a single locator. When a term $m^{[i:p]} \in \mathcal{D}$ is added to the set $\mathcal{C}$, its annotation is changed from $[i:p]$ to the one-element set $\{[i:p]\}$ and the term is written as $m^{\{[i:p]\}}$. Terms that are constructed from terms with locators are annotated with the set of locators consisting of the union of the locators of all terms used in the construction. I.e., the term $f(t_1^{j_1}, \ldots, t_k^{j_k})$ is annotated with the set $j_1 \cup \ldots \cup j_k$. The need for locators will become clear in the algorithm that computes the checks on the received messages.

Definition 2. *We say that a term $m^{[i:p]}$ can be* verified *with knowledge set M, if it can be constructed from M without using its subterms. Formally, $m^{[i:p]} \in \mathcal{D}$ is equal to some $m^L \in \mathcal{C}$, where $[i:p]$ is not a prefix of any element in L (and not equal to one). We define the function* `verifies`$(M, m^{[i:p]})$ *to return the witness m^L if it exists and $\perp$ otherwise.*

Note that the only terms that must be verified by an agent in a protocol execution are $\mathcal{D}$-terms.

While generating an agent's role script from an Alice&Bob specification we insert all possible checks that can be performed on received messages. In some cases, a received message can only be checked after further messages have been received, for instance in commitment schemes. In such a scheme, a principal receives an encrypted message first and the decryption key afterwards. The principal cannot check the first message until he receives the key. We use the set Γ to store messages for which checks may still be needed.

We start with the set $\Gamma = \emptyset$ and the set M equal to the agent's initial knowledge. All terms in the initial knowledge are annotated with [0]. Fresh nonces generated by the agent are also annotated with [0] and added to M, but not to Γ. We incrementally build the checks for messages in the order they are received. Let m be a message received in the i-th protocol step. We generate the possible checks by applying the algorithm `agent-checks`$(m^{[i]}, M, \Gamma)$ described

```
Input: m^[i], M, Γ.
Γ := Γ ∪ m^[i]
M := M ∪ m^[i]
M' := M
(*) ∀m^[j:p] ∈ M, ∀drule(l, q) ∈ drules:
      t^[j:p·q] := apply drule(l, q) to m^[j:p]
          if t^[j:p·q] ≠ ⊥ ∧ t^[j:p·q] ∉ M
              then Γ := Γ ∪ {t^[j:p·q]}, M' := M' ∪ {t^[j:p·q]}
if M' ≠ M then M := M' goto (*)

Γ' := ∅
∀γ ∈ Γ,
     if check(γ, M) = ⊥
          then Γ' := Γ' ∪ γ
          else print check(γ, M)
Γ := Γ'
return M, Γ
```

Fig. 4. Algorithm `agent-checks`($m^{[i]}$, M, Γ)

```
Input: m^[i:p], M.
m^L := verifies(M, m^[i:p])
if m^L ≠ ⊥
     then return m^[i:p] =? m^L
     else return ⊥
```

Fig. 5. Algorithm `check`($m^{[i:p]}$, M)

below on the message $m^{[i]}$, M, and Γ and afterwards update M and Γ with the algorithm's output. We define `agent-checks` in Figure 4. It uses the algorithm `check`, defined in Figure 5, that checks individual messages.

The algorithm `agent-checks` takes as input a message $m^{[i]}$, a set of unchecked terms Γ, and a knowledge set M. It first adds $m^{[i]}$ to both Γ and M. It then closes the knowledge set M with our knowledge closure algorithm modified to respect locators. Afterwards, it calls the `check` algorithm for all terms in the resulting set Γ. The result from `check` is used to add checks to the generated role script. Terms for which checks were added are removed from Γ.

The `check` algorithm creates checks for individual messages as follows. Its input is an annotated message $m^{[i:p]}$ (i.e., i-th protocol step, position p), and knowledge set M. If $m^{[i:p]}$ can be verified with M (see Definition 2) then `check` returns the corresponding check $m^{[i:p]} =^? m^L$, otherwise it returns $\bot$ to signify that no check is possible for this term with the given knowledge set.

Note that for the term $m^{[i:p]}$, the locator implicitly keeps track of how the term was derived from $m^{[i]}$. That way the check created in `check` compares the term derived from the received message $m^{[i]}$ with the term m^L constructed by the agent.

We briefly sketch why the checking algorithms are sound and complete. By construction, all checks $m^{[i:p]} =^? m^L$ generated by the `check` algorithm are computable. Both the left-hand term and the right-hand term are in the agent's knowledge. The checks are correct in the sense that $m^{[i:p]}$ is a subterm of a received message term and none of the subterms of m^L are derived from $m^{[i:p]}$. That is, the terms used to compute the left-hand and right-hand side have different origins. Finally, we generate all possible checks in the `agent-checks` algorithm since every subterm that can be derived from a received message is added to the set Γ. Each term in Γ that can be constructed from terms that have a different origin is checked against such a construction.

3.4 Putting It All Together

For a given protocol, we execute the above algorithms for each role. If the algorithms decide that the role is executable, they create a role script for the role's actions in the protocol. This includes the checks the role needs to make and warnings that are returned to the user. Essentially, everything left in Γ at the end of the protocol gives rise to a warning. Of course, if some term t cannot be checked and a warning is issued, then all terms that use t as a proper subterm cannot be checked either, and we therefore do not issue warnings for these terms.

The warnings are intended to inform the protocol designer about which message (sub-)terms the specified roles must accept as valid without having a way to check them. The protocol designer should then ensure that these unchecked parts are intentionally given as part of the specification. In particular, (i) they could be irrelevant (and should be dropped), or (ii) they are tickets that are just forwarded and checked by another role (and hence are there for a good reason), or (iii) they are important and protected under encryption with appropriate authentication. In this last case, the unchecked message part contains *new terms for this role* that the role must trust, and have confidence in, due to the overall protocol run. This could be, for example, a fresh key from a key distribution server, and generally speaking a term that is only received and not confirmed in any manner. Hence our approach not only creates explicit checks, but it determines which terms are not actually checked, and it calls the protocol designer's attention to this by issuing warnings.

Consider again the Kerberos authentication protocol previously given in Figure 2. By the conventions stated in Section 2.3, the protocol implicitly uses Equation (3) for the symmetric encryption and decryption of terms. Thus Σ contains the symbols `senc` and `sdec`. By Section 3.2 we first obtain the following two construction rules:

$$(x, y) \in \mathcal{C}^2 \vdash \texttt{senc}(x, y) \in \mathcal{C} \tag{5}$$

$$(x, y) \in \mathcal{C}^2 \vdash \texttt{sdec}(x, y) \in \mathcal{C}. \tag{6}$$

Next we apply the four step procedure in Section 3.2 to obtain a set of deconstruction rules for Equation (3).

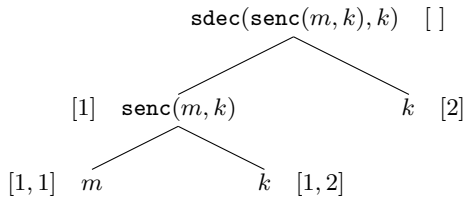

Fig. 6. Tree of subterms of $\mathtt{sdec}(\mathtt{senc}(m,k),k)$ and their positions.

1. We have $l = \mathtt{sdec}(\mathtt{senc}(m,k),k)$ and $r = m$. As illustrated in Figure 6, there is one position p in l such that $l|_p = r$, namely $p = [1,1]$.
2. By Figure 6, there is exactly one position that is above $p = [1,1]$ and not equal to $[\,]$, namely $[1]$, so $D_{[1,1]} = \{[1]\}$.
3. Figure 6 shows that $C_{[1]}$, the set of positions that have a sibling above or equal to $[1]$, is $\{[2]\}$.
4. Our deconstruction rule is therefore:

$$l|_{[1]} \in \mathcal{D} \wedge l|_{[2]} \in \mathcal{C} \vdash l|_{[1,1]} \in \mathcal{D},$$

that is

$$\mathtt{senc}(m,k) \in \mathcal{D} \wedge k \in \mathcal{C} \vdash m \in \mathcal{D}.$$

Similarly, we obtain three construction rules for the pairing operator `pair` and the two projections `fst` and `snd`, and two deconstruction rules for the two projections. We omit the details.

We now turn to the role scripts. For readability, we omit irrelevant terms from a role's knowledge. The three role scripts in the Kerberos protocol are then given as follows:

– Role script for role S

$$\begin{array}{lll} & S \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(C,S)^{[0]}, k(V,S)^{[0]} \\ \mathbf{1.} & C \to S: & (n).((C,V),n) \\ & S \text{ checks:} & (C,V)^{[1:[1]]} = (C^{[0]}, V^{[0]}) \\ & S \text{ checks:} & C^{[1:[1,1]]} = C^{[0]} \\ & S \text{ checks:} & V^{[1:[1,2]]} = V^{[0]} \\ & S \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(C,S)^{[0]}, k(V,S)^{[0]}, n^{[1:[2]]} \\ \mathbf{2.} & S \to C: & (k).\{k,V,n\}_{k(C,S)}, \{k,C\}_{k(V,S)}, \\ & S \text{ warns} : & n^{[1:[2]]} \end{array}$$

This role script is executable because S can generate the message in Step 2 from its knowledge shown above Step 2 by applying pairing and encryption. Our algorithm obtains the check after Step 1 as the first subterm (C,V) of the received pair $((C,V),n)$ is constructable from the initial knowledge. Also, both subterms of this pair, namely C and V, are constructable from the initial knowledge as well, so our algorithm adds an additional check each.

Note that our algorithm produces redundant checks here (those for $C^{[1:[1,1]]}$ and $V^{[1:[1,2]]}$), but these can be filtered out afterwards as they are subterms of the term $(C,V)^{[1:[1]]}$ checked against the same right-hand side. The only warning produced is for $n^{[1:[2]]}$, which is indeed a term that this principal cannot check anything about.

– Role script for role C

$$
\begin{array}{rll}
 & C \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(C,S)^{[0]} \\
\textbf{1.} & C \to S: & (n).C, V, n \\
\textbf{2.} & S \to C: & (k).\{k,(V,n)\}_{k(C,S)}, \{k,C\}_{k(V,S)} \\
 & C \text{ checks :} & (V,n)^{[2:[1,1,2]]} = (V^{[0]}, n^{[0]}) \\
 & C \text{ checks :} & V^{[2:[1,1,2,1]]} = V^{[0]} \\
 & C \text{ checks :} & n^{[2:[1,1,2,2]]} = n^{[0]} \\
 & C \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(C,S)^{[0]}, n^{[0]}, k^{[2:[1,1,1]]} \\
\textbf{3.} & C \to V: & (t,t').\{t,t'\}_k, \{k,C\}_{k(V,S)} \\
\textbf{4.} & V \to C: & \{t\}_k \\
 & C \text{ checks :} & (\{t\}_k)^{[4]} = \{t^{[0]}\}_{k^{[2:[1,1,1]]}} \\
 & C \text{ checks :} & t^{[4:[1]]} = t^{[0]} \\
 & C \text{ warns :} & k^{[2:[1,1,1]]}
\end{array}
$$

This role script is executable because C can generate the first message from its initial knowledge and the message in Step 3 from its knowledge shown above Step 3. The term k is obtained by applying Equation (3) to the first component of the pair received in Step 2 and picking the first element of the resulting pair. The only resulting warning is for the received key, which cannot be checked.

– Role script for role V

$$
\begin{array}{rll}
 & V \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(V,S)^{[0]} \\
\textbf{3.} & C \to V: & (t,t').\{t,t'\}_k, \{k,C\}_{k(V,S)} \\
 & V \text{ checks :} & C^{[3:[2,1,2]]} = C^{[0]} \\
 & V \text{ knows:} & S^{[0]}, C^{[0]}, V^{[0]}, k(V,S)^{[0]}, k^{[3:[2,1,1]]}, t^{[3:[1,1,1]]}, t'^{[3:[1,1,2]]} \\
\textbf{4.} & V \to C: & \{t\}_k \\
 & V \text{ warns :} & k^{[3:[2,1,1]]}, t^{[3:[1,1,1]]}, t'^{[3:[1,1,2]]}
\end{array}
$$

This role script is executable because V can generate the message $\{t\}_k$ from its knowledge. The resulting warning is about the three terms for which no check can be included: the key k and the two nonces t and t'.

4 Automated Translation to Tamarin

An instance of our Alice&Bob translation to a tool-supported protocol specification language is described in more detail in [18]. The translation goes from Alice&Bob via an *intermediate representation* format to Tamarin's input language. The intermediate representation is functionally the same as the role

scripts described in this paper, with minor syntactic differences. TAMARIN uses the tool-generated input to analyze the given protocol.

On loading a protocol theory, TAMARIN detects when a protocol rule is not executable; however our automatic translation only produces theories that meet the executability requirement. TAMARIN supports user-specified subterm-convergent equational theories. Our translation therefore simply copies the equational theory with some minor syntactic changes. Checks on received messages are implemented by pattern matching on the premises of rules. The security goals of secrecy, non-injective agreement, and injective agreement are translated into their canonical definition used when specifying protocols for TAMARIN.

More detail and further examples are available in [18] and the tool is available at the webpage [3]. Due to space constraints, we do not list the TAMARIN specification for our running example produced by the translation.

There is also a prototype implementation of an explicit check generator, following our algorithms described above. It is available at [3].

5 Related Work

We will first discuss other research related to Alice&Bob notation. Afterwards we consider different tools' input languages, which are the target languages for our translation effort. Finally, we discuss other proposed translation mechanisms.

Formalization of Alice&Bob Notation. Alice&Bob notation, while intuitive, suffers from ambiguities and imprecision as shown in Section 2.1. To clarify what protocol specification notation actually means, Caleiro et al. [8,9] and Mödersheim [26] have investigated the semantics of Alice&Bob notation.

Caleiro et al. work with a fixed message model and consider how principals' knowledge increases during a protocol run as principals receive messages. Moreover, they provide an operational semantics, based on the spi calculus, that makes explicit the actions that a principal must execute. The key aspect of the operational semantics is that it provides detailed checks to be performed by principals to ensure there was no adversary involvement. Our semantics of Alice&Bob protocol specifications are based on this work.

In contrast to Caleiro et al.'s semantics, which is based on a fixed message model, Mödersheim gives a formalization of Alice&Bob notation that is defined over an arbitrary algebraic theory. However, his method does not directly yield the actions that must be taken by honest principals.

Input Languages. Each automated security protocol verification tool defines its own input language. Most of these languages look rather different from Alice&Bob notation. However, their underlying concepts are often similar to the core idea of Alice&Bob notation: communication is modeled by specifying the messages that are sent and received by principals participating in a protocol run. Most input languages however do not explicitly pair the sender and the receiver as is done in Alice&Bob notation.

In Maude-NPA [15] protocols are specified by defining *strands*, which are similar to the *roles* used in our work. A strand specifies a sequence of sending and receiving messages, from one participant's point of view. Using Maude's [11] unification capabilities, Maude-NPA then reasons modulo equational theories.

Similarly, protocols in Scyther [13] are specified by explicitly stating which actions (sending, receiving, generation of fresh numbers, and claims of security properties) must be taken by principals. Scyther-proof [23] is a tool based on a proof-generating variant [24] of the verification theory underlying Scyther. Its input language uses proper Alice&Bob-style notation for specifying protocols.

In ProVerif [7] protocols are specified in the applied pi calculus as the parallel compositions of processes that correspond to roles in Alice&Bob notation. Checks on received messages must be explicitly stated.

Tamarin's [29] input language is based on specifying rewriting rules for multisets of so-called facts. State is usually expressed with the help of user-defined facts and communication by the predefined `In` and `Out` facts, which represent sending and receiving actions. Hence, Tamarin also works by stating the send and receive actions of principals.

Even though all of the tools' input languages have aspects in common with Alice&Bob notation, they all heavily rely on the specification of additional information, such as algebraic properties and typing rules, which must be stated explicitly. Mödersheim uses an elegant Alice&Bob-style language called AnB [26] where the algebraic properties of the message model are assumed to be fixed, and consequently need not be included in the protocol specification itself. In our work, we fuse the different approaches into an input language that has a pre-defined message model, similar to [26], but which is extensible with user-specifiable subterm-convergent equations while leaving the adversary model unspecified.

Existing Translations. There are two steps that a translation from an Alice&Bob input language into a tool-specific language must perform. The first is to verify that a given Alice&Bob specification is executable and the second is to extract the security checks that a role must perform on received messages. It is in these two steps that existing translations differ. Executability is important, as otherwise protocol participants cannot carry out their steps and run the protocol. Non-executability indicates either a mistake in the protocol or its formalization, for example, a missing setup assumption. Checks on incoming messages are also important, for example, to ensure that the message authentication code a participant received refers to the message actually received.

Chevalier et al. [10] translate protocol narrations to strand-like role scripts that are annotated with explicit unifiability conditions. Their protocol narrations are similar to our Alice&Bob input, i.e., they specify the messages exchanged as well as the sending and receiving agent, and the initial knowledge of each agent. They verify executability during the translation to role scripts. Their security checks are such that each participant verifies received messages as far as possible. Namely, a message item that is reused in the description must be the same for the receiver in all incoming messages. This imposes checks that are

practically infeasible for agents to perform and we give an example in Section 3.3, Example 1. For these cases, we do not add such infeasible checks; instead we warn the protocol designer that something might not be working as intended due to the inherent imprecision of such protocol narrations or Alice&Bob specifications. The output of their translation is not directly analyzable by any existing tool, unlike ours, which can directly output descriptions that can be analyzed by Tamarin, in addition to producing role scripts.

Another alternative translation is based on endpoint projections [22]. The protocol, given in a language close to Alice&Bob notation, is transformed in multiple steps to a role script-like language. Along the way, the implicit assumptions in the input are made explicit by making choices. The endpoint projection must, in particular, deal with the asymmetries introduced by security protocols, for example, the receiver might not (yet) know the key being used in an encryption. Making these choices to get to an explicit presentation is similar to what we do. The main restrictions in their work, compared to ours, is that their translation cannot handle equational theories beyond equations formalizing symmetric and asymmetric encryption and it does not allow principals to transmit two or more messages in a row without receiving a message in between.

The *Common Authentication Protocol Specification Language* CAPSL [14] uses message list input similar to our Alice&Bob specifications, extended with Casper's % operator [21] to indicate receiver patterns. The use of these extended patterns makes protocol specification more cumbersome, but it substantially simplifies the problem of determining which checks should be made when receiving messages, which we do in our work without resorting to such patterns. CAPSL translates the input message lists into an intermediate language CIL, given in multiset rewriting. Subsequent translations from CIL to different analysis tools, such as Maude, have also been carried out [14].

6 Conclusions

We have presented an analysis of Alice&Bob style protocol specifications, yielding a translation into an intermediate role script-like format that handles executability concerns and generates appropriate checks for correct message reception. We have also implemented a further translation of this intermediate format to the input of Tamarin, a cryptographic protocol verification tool that uses multiset rewriting for protocol specifications. This translation is automated and allows Alice&Bob protocol specifications to be analyzed and verified with Tamarin.

Alice&Bob notation is simple and can be used by novices. Indeed we believe that our work could help to teach undergraduate students about protocol specification and analysis in a formal methods course, with some caveats. The students can specify protocols nicely using Alice&Bob notation, and, when the back-end verification succeeds, they have proven the security property. However, when verification fails, the tool's counter-example is still in a format that is not very meaningful for students. To make this viable for teaching, one would need to

define and implement a back-translation from the tool's output representation to Alice&Bob like syntax that is easier to understand. This back-translation should benefit from knowledge gained in the initial translation from Alice&Bob notation to the tool's input. In particular, for TAMARIN, this counter-example output is in the form of a constraint system of dependency graphs. We are investigating the conversion of these back to Alice&Bob notation as future work.

References

1. Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV*, pages 281–285, 2005.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. David Basin, Michel Keller, Saša Radomirović, and Ralf Sasse. Alice&Bob protocols. Available at `http://www.infsec.ethz.ch/research/software/anb.html`.
4. David A. Basin, Cas Cremers, and Simon Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. *Journal of Computer Security*, 21(6):817–846, 2013.
5. David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 382–393. ACM, 2014.
6. B. Blanchet. Proverif automatic cryptographic protocol verifier user manual. *CNRS, Departement dInformatique, Ecole Normale Superieure, Paris*, 2005.
7. Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE, 2001.
8. Carlos Caleiro, Luca Viganò, and David Basin. Deconstructing Alice and Bob. *Electronic Notes in Theoretical Computer Science (Proceedings of the Workshop on Automated Reasoning for Security Protocol Analysis, ARSPA 2005)*, 135(1):3–22, 2005.
9. Carlos Caleiro, Luca Viganò, and David A. Basin. On the Semantics of Alice&Bob Specifications of Security Protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.
10. Yannick Chevalier and Michaël Rusinowitch. Compiling and securing cryptographic protocols. *Information Processing Letters*, 110(3):116–122, 2010.
11. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
12. Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *RTA*, volume 3467 of *LNCS*, pages 294–307. Springer, 2005.
13. Cas J.F. Cremers. Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement. In *ACM Conference on Computer and Communications Security (CCS)*, pages 119–128. ACM, 2008.

14. Grit Denker and Jonathan K. Millen. CAPSL intermediate language. In *Proceedings of FMSP'99*, 1999. Available at `http://www.csl.sri.com/users/millen/capsl/`.
15. Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.
16. Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.
17. F. J. Thayer Fabrega, J. Herzog, and J. Guttman. Strand Spaces: What Makes a Security Protocol Correct? *Journal of Computer Security*, 7:191–230, 1999.
18. Michel Keller. Converting Alice&Bob Protocol Specifications to Tamarin. Bachelor's thesis, ETH Zurich, 2014. Available at `http://www.infsec.ethz.ch/research/software/anb.html`.
19. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, pages 147–166, 1996.
20. Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 31–, Washington, DC, USA, 1997. IEEE Computer Society.
21. Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1):53–84, 1998.
22. Jay A. McCarthy and Shriram Krishnamurthi. Cryptographic protocol explication and end-point projection. In *ESORICS 2008*, volume 5283 of *LNCS*, pages 533–547. Springer, 2008.
23. Simon Meier. GitHub Repository of scyther-proof Project. `https://github.com/meiersi/scyther-proof`.
24. Simon Meier, Cas J. F. Cremers, and David A. Basin. Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs. In *CSF*, pages 231–245. IEEE Computer Society, 2010.
25. Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, CAV 2013*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.
26. Sebastian Mödersheim. Algebraic Properties in Alice and Bob Notation. In *ARES*, pages 433–440. IEEE Computer Society, 2009.
27. B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.
28. Benedikt Schmidt. *Formal Analysis of Key Exchange Protocols and Physical Protocols*. PhD dissertation, ETH Zurich, 2012.
29. Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.
30. Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David A. Basin. Automated verification of group key agreement protocols. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 179–194. IEEE Computer Society, 2014.